Notes on CRC (Cyclic Redundancy Check)

Brian Mearns

2013 July 10

Abstract

The following is a transcription of some notes I've put together on CRCs, cyclic redundancy checks. CRCs are used for error detection and integrity validation in digital data. Much of this is my own summary or rehash of the very excellent document "A Painless Guide to CRC Error Detection Algorithms," written by Ross N. Williams, and available at http://www.ross.net/crc/download/crc_v3.txt. While Williams may be an expert on CRCs, I am not: I make no guarantees as to the accuracy of anything in this document.

1 CRC

Basic idea: the data is a binary polynomial. Divide this polynomial by another, the remainder is the CRC check value.

Polynomial example:

$$\text{Binary Data} : 101110$$
$$\text{Polynomial} : x^{32} + x^{8} + x^{4} + x^{2}$$

If M is the message to check, then mathematically, we want to construct a value C, based on M, which is evenly divisible by a specific polynomial, G (called the *generator polynomial*). Since all values are binary polynomials, division is done using binary arithmetic *with no carriers.*

For instance, suppose $\frac{M}{G} = Q \text{ rem } R_0$, which is to say that M divided by G gives a quotient Q and a remainder R_0. Equivalently: $\frac{M}{G} = Q + \frac{R_0}{G}$, or $M = GQ + R_0$.

From this, we can see that $M - R_0$ will be evenly divisible by G:

$$\begin{aligned} M &= GQ + R_0 \\ M - R_0 &= GQ + R_0 - R_0 \\ M - R_0 &= GQ \\ \frac{M - R_0}{G} &= Q \text{ rem } 0 \end{aligned}$$

Therefore if we transmit $M - R_0$, the receiver can divide by G, and if it divides evenly (no remainder), we assume there was no error. If there is a non-zero remainder, there must have been an error.

This is the basic premise, for error detection with CRC. However, M is not recoverably from $M - R_0$ unless we know R_0, which the receiver does not. Therefore, we can detect (some) transmission errors, but we haven't actually transmitted the message.

What we would like to do is send both M and R_0, then the receiver can do the subtraction themselves and see if $M - R_0$ is evenly divisible by G.

We can do this by concatenating M and R_0 together. Mathematically, this means shifting M up by however many bits are in R_0 and then adding R_0 to the result. But once again, if the receiver doesn't know R_0, he doesn't know how many bits it has, and therefore cannot reconstruct M.

However, we do know that R_0 is strictly less than G (since it was the remainder of division by G), which means it has no more bits than G. Therefore, we can agree that we will always treat R_0 as have the same number of bits as G, and simply zero fill the upper bits as needed.

In summary:

- Assume G has W bits.
- Multiply M by 2^W, which is the same as appending W bits with value 0 to the end of M. Call this elongated message N ($N = M2^W$). (Note, in the original notes, I called this E instead of N, but that's confusing because E is usually used to designate an error string).
- Divide N by G using binary polynomial arithmetic: $\frac{N}{G} = Q \text{ rem } R$
- The quotient Q is ignored.
- Subtract the remainder R from N to get our composite message *with* check bits. We call this composite message C: $C = N - R = (M2^W) - R$

- Transmit C.

Note that we have made an important change here, from R_0 to R. This is because R_0 was the remainder of M/Q, and now R is the remainder of N/G, or in otherwords, $\left(M2^W\right)/G$. From here on out, we will only be considering R, not R_0. They have different values but serve the same conceptually purpose. Al we have done is take our message M and created a new message N from which we can subtract a W-bit remainder (R) without loosing any information about N. This is because we know the bottom W bits of N will always be 0.

Also note that in the arithmetic we're using, addition and subtraction are the same, and they are both the same as bit-wise exclusive OR. This is because we're not using carries and so $1 + 0 = 1$, $0 - 1 = 1$, and $1 + 1 = 1 - 1 = 0$. Therefore $N - R$ is equal to $N + R$, and since N is simply M shifted up by W bits, we have effectively appended the W-bit remainder R to the message M. We will use the notation $C = M \| R$ to indicate that C is composed of R appended to M.

For clarity, assume that the message *received* by the receiver is $\widetilde{C}$, which may or may not be equal to the transmitted message C. This of course is the reason we need error detection in the first place, because errors could occur during transmission which convert C in to $\widetilde{C}$. The tilde accent over a symbol is generically used in these notes to indicate that it may include errors, so the notation $\widetilde{C}$ can be read as "C with possible errors". For brevity, I usually read it as "C-sharp" or "C-bang".

Since we are dealing with binary data, any error that may occur can be regarded as a bit string which is added to the original data. For instance, we might write $\widetilde{C} = C + e_C$, where e_C is called the *error string*. Remember that all arithmetic is done using binary polynomial arithmetic, which means binary arithmetic without carries. This means that addition of two bit strings is actually just an XOR operation. Therefore, any bit that is set in the error string indicates that the corresponding bit in the original message has been corrupted by an error, causing it to have the opposite value in the received data.

For instance, if $C = 10110010$ and $e_C = 00100011$, then the received message with error is $\widetilde{C} = 10010001$. If there are no errors, then all the bits in the error string are 0 and the received message is equal to the transmitted message.

So we have constructed and transmitted $C = M \| R$, and we can say the receiver has received $\widetilde{C} = \widetilde{M} \| \widetilde{R}$. In otherwords, the receiver can likewise break up the received message into two parts: $\widetilde{M}$, which is the original message possibly with errors, and $\widetilde{R}$, which is the original remainder possibly with errors.

To determine if there are errors, the receiver has a few options. They can perform the division $\frac{\widetilde{M}}{G}$ using the agreed upon generator polynomial G, and check if the remainder of that division is equal to the received check bits $\widetilde{R}$. Alternatively, they can perform the subtraction $\widetilde{M} - \widetilde{R}$ and divide the difference by G, verifying that the remainder is 0.

In either event, if the remainder does not match what was expected ($\widetilde{R}$ in the first case, 0 in the second case), the receiver will know that an error has occurred during transmission and $\widetilde{C} \neq C$. Note that in either case, a mismatch could be cause either by an error in the message $\widetilde{M}$, or by an error in the check bits $\widetilde{R}$ (or errors in both). The receiver has no way of knowing, but in either case a transmission error has occured and the received message must be assumed to be incorrect.

It is also certainly possible that an error could occur during transmission which is not detected by the algorithm. Remember that ultimately, the check just comes down to verifying a remainder against an expected value. If an error occurs which causes the erroneous message to yield the correct remainder, then the error will be undetected. These notes will not go into details about what errors can be detected, but suffice it to say that any error strings which are no longer than the number of bits in the remainder are guaranteed to be detected. In this regard, the length of an error string is the number of bits from the first set bit to the last set bit, inclusive. For instance, the error string 0001010001000 has a length of 7.

We saw how the receiver can decompose the received message $\widetilde{C}$ into components $\widetilde{M}$ and $\widetilde{R}$ to perform the data verification, but there is a simpler and more elegant solution as well. Since R is the remainder of $M2^W/G$, we know that $\big(M2^W - R\big)$ will divide evenly by G (just like how we saw $(M - R_0)$ divides evenly by G). Also recall that $M2^W - R$ is the same as $M \| R$, which is simply the transmitted message C. Therefore, C as a whole should divide evenly by G. To check for errors in the received message, the receiver can simply divide the entire received message $\widetilde{C}$ by G and check for a remainder. A non-zero remainder indicates that $\widetilde{C} \neq C$ and therefore an error has ocurred during transmission.

Lastly, to get the original message from the received message (assuming no errors), the receiver can simply cut off the last W bits from $\widetilde{C}$, which is the appended remainder $\widetilde{R}$. The bits prior to this are the transmitted message $\widetilde{M}$.

2 Division

In this arithmetic, two bit strings with the same number of significant bits cannot be compared for size because addition and subtraction are the same. For instance:

$$1011 + 101 = 1110$$
$$1011 - 101 = 1110$$

From this, you can see that it is not meaningful to say that either 1011 is bigger than 1110, or vice-versa.

Therefore, two strings with the same number of significant bits can always be divided each into the other with a non-zero quotient. For example:

$$1110 = 1\,(1011) + 101 \rightarrow \frac{1110}{1011} = 1 \text{ rem } 101$$

And inverted:

$$1011 = 1\,(1110) + 101 \rightarrow \frac{1011}{1110} = 1 \text{ rem } 101$$

However, a string with fewer significant bits is always strictly less than one with more significant bits. Therefore, a number with more significant bits divides a number with fewer significant bits zero times (with a remainder).

The point is, we can do this type of division using long division. We will be doing the division N/G, so the N is the dividend and G is the divisor (that's on the transmitting side, on the receiving side, it would actually be $\widetilde{N}/G$, but of course the division works the same). The string G has W bits, and since $N = M2^W$, we know that N has at least W bits as well, generally, N will have much more than W bits.

To do the long division, we consider the dividend N in groups of W bits at a time. This is just like in ordinary decimal long division, if the divisor has D digits, then you consider the dividend D digits at a time. The important difference here, as we just saw, is that if that group of W bits in the dividend has the most significant bit set, then it has W significant bits and so the W-bit dividend (G) divides into it once (possibly with a remainder). In other words, the quotient bit will be 1. On the other hand, if the W-bit dividend group does not have the most significant bit set, then it has fewer than W significant bits and the divisor therefore does not go into any whole number of times, so the quotient bit will be 0.

The only other note for doing the division is that the subtraction used to find the remainder in each step is done using the same binary polynomial arithmetic, meaning that subtraction is simply a bit-wise XOR, and there are no carries or borrows.

Here is an example. Note that this is just a generic example of doing this type of division. You can tell that the dividend is not an "N" value because it does not have the correct number of least significant zero bits (N would have W such bits, and here W is 5).

To begin with, we note that the divisor (10011) has five bits, so if we try to divide it into any of the first four bits, it will have a quotient of 0, so we can skip those. (Just like in ordinary decimal long division: if you're dividing 156201 by 1392, you would skip trying to divide 1, 15, or 156 by the divisor, and go straight to dividing 1562 by 1392).

The first five bits of the dividend (the most significant) are 11010, and since that has 5 significant bits, we know that our 5-bit divisor will go into it once, possibly with a remainder. So we write a quotient of 1 above this group, multiply the divisor by this quotient, and copy it below the corresponding bits in the dividend. Note that the nice thing about doing binary division is that the quotients are always either 1 or 0 (1 if there are the same number of significant bits as in the divisor, 0 otherwise), so multiplying the divisor by this quotient digit will always yield either the divisor itself, or 0.

```
              1
        _____________
10011  )1101011011000
        10011
```

As in ordinary long division, we subtract this product from the bits above it. The difference is that this is bit-wise binary subtraction, so it's simply an XOR.

```
              1
        _____________
10011  )1101011011000
        10011
        -----
        01001
```

Notice that the resulting difference will always have at most $W - 1$ significant bits. This is because either the dividend had the same number of signficant bits as the divisor, in which case both operands of the XOR will have their most signficant bits set producing a 0 in the corresponding bit in the result; or the dividend had fewer signficant bits than the divisor in which case the most significant bit of the dividend is necessarily 0 and the second operand in the XOR is all zeroes. Of course, this is the same as in ordinary long division, where we know that the result of this subtraction should be less than the divisor, otherwise we picked a quotient that was too small.

The next step, just as in ordinary long division, is to bring down the next bit and treat the concatenated value as the new dividend.

```
            1
        -------------
10011  )1101011011000
        10011
        -----
        010011
```

Now we're dividing 10011 by 10011 (just a coincidence), which of course has a quotient of 1. So we write that quotient bit, copy the divisor down (i.e., multiply it by the quotient, one), and perform the subtraction/XOR, then bring down the next bit to form the new dividend.

```
            11
        -------------
10011  )1101011011000
        10011
        -----
        010011
         10011
         -----
         000001
```

The rest of the division is shown below. The bits of the dividend are colored to show how they are brought down.

```
            110000101
        -------------
10011  )1101011011000
        10011
        -----
        010011
         10011
         -----
         000001
          00000
          -----
          000010
           00000
           -----
           000101
            00000
            -----
            001011
             00000
             -----
             010110
              10011
              -----
              001010
               00000
               -----
               010100
                10011
                -----
                00111
```

The process ends when there are no more bits to bring down in the divisor. Whatever is left over at this point is the remainder: in this case it is the 00111 shown at the very bottom of the figure. The quotient is shown above the divisor: 110000101. Note that for CRC, we don't actually care about the quotient.

In general, you can check yourself by multiplying the quotient by the divisor, and adding in the remainder. Remember, however, that this is binary polynominal arithmetic. In the problem above, the quotient represents the polynomial $x^8 + x^7 + x^2 + 1$, the divisor is $x^4 + x + 1$, and the remainder is x^2+x+1. So our dividend, $x^{12}+x^{11}+x^9+x^7+x^6+x^4+x^3$, should be equal to:

$$\left(x^8 + x^7 + x^2 + 1\right)\left(x^4 + x + 1\right) + \left(x^2 + x + 1\right)$$

You can distribute multiplication by the quotient to each addend in the divisor as follows:

$$\begin{aligned}
\left(x^8 + x^7 + x^2 + 1\right)\left(x^4 + x + 1\right) \\
&= x^4\left(x^8 + x^7 + x^2 + 1\right) \\
&\quad + x\left(x^8 + x^7 + x^2 + 1\right) \\
&\quad + 1\left(x^8 + x^7 + x^2 + 1\right) \\
&= x^{12} + x^{11} + x^6 + x^4 \\
&\quad + x^9 + x^8 + x^3 + x \\
&\quad + x^8 + x^7 + x^2 + 1 \\
&= x^{12} + x^{11} + x^9 + 2x^8 + x^7 + x^6 + x^4 + x^3 + x^2 + x + 1 \\
&= x^{12} + x^{11} + x^9 + x^7 + x^6 + x^4 + x^3 + x^2 + x + 1
\end{aligned}$$

Note that in the second to last line above, we cheated a little by writing $2x^8$. This was done for clarity, but since this is binary arithmetic, there is no 2. In fact, as we've seen through out, arithemtic is done without carries, or in other words, in modulo-2. Therefore 2 is actually the same as 0, meaning when we added the two x^8s together, they canceled each other out. This is shown in the last line.

Lastly, we add to this the remainder, $x^2 + x + 1$. Notice that we already have each of these terms so, just like the x^8, they will simply cancel out, leaving us with a value of

$$x^{12} + x^{11} + x^9 + x^7 + x^6 + x^4 + x^3$$

In binary representation, this is 1101011011000, which matches our dividend, as it must.

2.1 Division for CRC

Since we don't actually care about the quotient for CRC, take a look back at the process of doing the long division, paying particular attention to the remainders (i.e., the difference taken at each step, along with the next bit brought down). Remember that the remainder from one step becomes the dividend for the next step. Therefore, if the remainder has the same number of significant bits as the divisor (i.e., it has W significant bits), then the quotient for this step will be 1, which means we are going to end up subtracting (XORing) the divisor from the current remainder in the next step. On ther other hand, if the remainder has fewer significant bits than the divisor, the quotient for this step will be 0, and the remainder will be unchanged in the next step (ignoring the next bit to be brought down). Note that the remainder cannot have more signficant bits than the divisor, that would indicate that the previous quotient was wrong.

Either way, we will then shift the next bit from the original dividiend onto the least signficant end of the remainder.

Now, if the divisor is G and has W significant bits, the dividend is an array of bits N, and the remainder is R, we have:

```
R=0;
for(bit in N) {
    //Check if R has W signficant bits.
    if ((R >> (W-1)) == 1) {
        //bit-wise binary subtraction.
        R = R ^ G;
    }
    //Shift in the next bit from dividend (N).
    R = (R << 1) | bit;
```

That's the basic, straightforward, algorithm for getting the remainder for CRC. Remember that for CRC, we need to shift the message up by W bits before doing the division.

3 Table Based Implementation

Consider what happens as we process 2 bits of the message. Imaginer we're using a 5-bit polynomial G, and out remainder register currently looks like this:

a_4	a_3	a_2	a_1	a_0

When we're ready for the next bit from the dividend, call it m_i, it will be the most signficant bit from the remainder register, a_4 in this case, which will determine whether or not we XOR G into the register:

$$\begin{array}{r}
\begin{array}{|c|c|c|c|c|}\hline a_3 & a_2 & a_1 & a_0 & m_i \\ \hline\end{array} \\
+\left(a_4 * \begin{array}{|c|c|c|c|c|}\hline g_4 & g_3 & g_2 & g_1 & g_0 \\ \hline\end{array}\right) \\ \hline
\begin{array}{|c|c|c|c|c|}\hline b_4 & b_3 & b_2 & b_1 & b_0 \\ \hline\end{array}
\end{array}$$

Notice that the resulting bit values, b_i, are defined as follows:

$$\begin{aligned}
b_4 &= a_3 + (a_4 g_4) \\
b_3 &= a_2 + (a_4 g_3) \\
b_2 &= a_1 + (a_4 g_2) \\
b_1 &= a_0 + (a_4 g_1) \\
b_0 &= m_i + (a_4 g_0)
\end{aligned}$$

With the next bit, it will be $b_4 = a_3 + (a_4 g_4)$ which will determine whether or not we do the XOR:

$$\begin{array}{r}
\begin{array}{|c|c|c|c|c|}\hline b_3 & b_2 & b_1 & b_0 & m_{i+1} \\ \hline\end{array} \\
+\left(b_4 * \begin{array}{|c|c|c|c|c|}\hline g_4 & g_3 & g_2 & g_1 & g_0 \\ \hline\end{array}\right) \\ \hline
\begin{array}{|c|c|c|c|c|}\hline c_4 & c_3 & c_2 & c_1 & c_0 \\ \hline\end{array}
\end{array}$$

Our new result bits, c_i, are defined as follows:

$$\begin{aligned}
c_4 &= b_3 + (b_4 g_4) & &= a_2 + (a_4 g_3) + ((a_3 + (a_4 g_4))\, g_4) \\
c_3 &= b_2 + (b_4 g_3) & &= a_1 + (a_4 g_2) + ((a_3 + (a_4 g_4))\, g_3) \\
c_2 &= b_1 + (b_4 g_2) & &= a_0 + (a_4 g_1) + ((a_3 + (a_4 g_4))\, g_2) \\
c_1 &= b_0 + (b_4 g_1) & &= m_i + (a_4 g_0) + ((a_3 + (a_4 g_4))\, g_1) \\
c_0 &= m_{i+1} + (b_4 g_0) & &= m_{i+1} + ((a_3 + (a_4 g_4))\, g_0)
\end{aligned}$$

So we can see that after processing two message (dividend) bits, each bit in our remainder register has values of the form:

$$c_j = x_j + (a_4 g_{j-1}) + ((a_3 + (a_4 g_4))\, g_j) \qquad (1)$$

where the vector $X = \langle x_0 \dots x_4 \rangle$ is defined as $\langle m_{i+1}, m_i, a_0, a_1, a_2 \rangle$, and we define g_{-1} as 0 (used in c_0).

This can be further simplified as

$$c_j = x_j + t_j \qquad (2)$$

where t_j is only dependent on G and on the top two bits of the starting value in the remainder regsiter, a_3 and a_4.

So at this point, we have:

$$\begin{array}{r}
\boxed{a_2 \mid a_1 \mid a_0 \mid m_i \mid m_{i+1}} \\
+\boxed{t_4 \mid t_3 \mid t_2 \mid t_1 \mid t_0} \\
\hline
\boxed{c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0}
\end{array}$$

Recall that the bitstring $T = \langle t_0, \ldots, t_4 \rangle$ is a function only of a_4, a_3, and the generator polynomial G. That means we can precompute T for all possible values of a_4 and a_3 for a given G. Once we have that, all we need to do is shift two of message into the remainder register from the right, and XOR with the value $T(a_3, a_4, G)$.

A nicer way to write this is $T_G(a_3, a_4)$, which makes T_G a four element look up table specific to the generator polynomial G. A value is pulled from the table using the binary value of the bitstring $\langle a_4, a_3 \rangle$ as an index (in other words, $2{*}a_4{+}a_3$), and the value is then XOR'd into the remainder register.

The reason T_G is only indexed using the top two bits of the register (a_3 and a_4) is because we looked at shifting two message bits into the register before XORing the value from T_G. If we had shifted in a third bit (m_{i+2}) before doing the XOR, then a_2 would have shifted off the register and affected the XOR value as well. In this case, we would have:

$$\begin{array}{r}
\boxed{a_1 \mid a_0 \mid m_i \mid m_{i+1} \mid m_{i+2}} \\
+\boxed{\quad T_G(a_2, a_3, a_4) \quad} \\
\hline
\boxed{d_4 \mid d_3 \mid d_2 \mid d_1 \mid d_0}
\end{array}$$

Of course, T_G can be made for various numbers of bits. T_G for S bits will naturally have 2^S entries, each entry is the same size as the polynomial G. The bigger S is, the faster to can process the message (S bits at a time), but the bigger the table needs to be. That means you need more memory and it takes longer to generate the table.

The limit for S is that S cannot exceed W, the number of bits in the polynomial G. If you try to shift more than W bits at a time into the register, then the leading messages bits will be shifted all the way out and not end up affecting the XOR value.

3.1 Computing the Table

Recall that we initially setup the premise of using a value from a look up table simply by coming up with a generic expression for the result after two iterations in equation 2, repeated here for convenience:

$$c_j = x_j + t_j \tag{2 rpt}$$

Here c_j is the resulting bit we're after, and x_j is from the bitstring $X = \langle a_2, a_1, a_0, m_i, m_{i+1} \rangle$ (from most significant to least significant), where a_2, a_1, and a_0 are the low order bits of the initial remainder value, and m_i and m_{i+1} are message bits. The new item introduced in this equation was the bit t_j which we used as a place holder to simplify several terms of the generic equation 1 which came directly from performing the algorithm, once again repeated here for convenience:

$$c_j = x_j + (a_4 g_{j-1}) + ((a_3 + (a_4 g_4))\, g_j) \tag{1 rpt}$$

If we preload our remainder register with the bitstring $\langle a_4, a_3, 0, 0, 0 \rangle$, and set the message to be all zeros, then we have $x_j = 0$ for all j, by definition of the bitstring X.

Plugging this into equation 1, we find that after two iterations we will produce a register value Y whose bits have values according to:

$$y_j = 0 + (a_4 g_{j-1}) + ((a_3 + (a_4 g_4))\, g_j)$$

It is clear that the values y_j are exactly equal to the values t_j in equation 2, thus the resulting register value Y is the value we want to store in the look up table, $T_G(a_3, a_4)$.

We have shown this example of generating a look up table for $S = 2$, but it works the same way for any value of $S \leq W$.

To summarize, the values for each entry in the lookup table are computed by filling the top S bits of the remainder register with the index bits (most significant index bit in the most significant register bit), and filling the rest of the register with zeroes, then performing S iterations of the normal division/remainder algorithm using a message/dividend consisting entirely of zeroes. The resulting remainder value is the value to store in the table.

3.2 Using the Table

Just to summarize how to use this implementation. You would need to agree on the generator polynomial in advance. There are lots of different commonly used polynomials out there, some have certain strengths and weaknesses.

Once you've agreed on the polynomial, you can generate and store your table. Exactly how you store your table is up to you, and is independent of the other communicating party, as long as you're using the same polynomial and bit-ordering conventions. For instance, the transmitter might generate a table with 256 entries and process data 8-bits at a time while the receiver generates a table with only 32 entries and processes it 5 bits at a time. You could also choose not to generate a table at all, and just use the standard algorithm. The whole point is that all these different methods implement the same algorithm, so they can be used interchangeably. What varies is the resource requirements.

As for generating the table, it will very frequently be calculated "off-line" or "out-of-band" and stored as constant read-only data in your implementation. This is the most time efficient, but makes your implementation larger (in software and firmware, this will consume the ROM resources). Alternatively, you could generate the table inside your implementation when it is needed. This will generally consume RAM resources but not ROM resources, making your implementation smaller than if it contained the table itself. The trade off, of course, is that your implementation needs to take the time to generate the table before it can process any data. If you have a large amount of data to process, this may be worth while, but if you have a small amount of data to process, it may take more time to generate the table than it would to do a straight foward "naive" implementation of the algorithm.

The size of the table is likewise a trade off. A single table operation involves pulling a number out of the table and XORing it into a register. For the most part, this will take a fixed amount of time regardless of the table size, so the more bits you can process in a single operation, the faster you can process the data. However, processing more bits at a time requires a larger table, which means more memory resources (either RAM or ROM) and more time to generate the table (which may not matter if it's done out-of-band).

So you've got your table and your message to checksum. If you're the transmitter, then you will want to append W more bits to the end of your message, all with value value (this is the process of generating N from M, using the notation from above). You initialize your remainder register to 0. Then, you shift the S most significant bits out of this register and keep them as an index value I. Meanwhile, you shift *in* the next S bits of your message, *from most significant to least significant* so that the last W bits that you process will be the zeroes that you appended. With the register thus shifted and loaded, you use I as an index into your table, and XOR that value from the table into the register. Repeat untill all data has been processed.

In the end, you will have a remainder value left in the register. Append this as a W-bit string to your *original* message (or alternatively, use it to replace the W least significant bits that you appended to create N), and transmit the resulting bit string.

The receiver side works similarly, except there is no need to pad of extend the message. You have already received the bitstring representing the polynomial $(\widetilde{N - R})$, so if there were no errors, it should be a multiply of G. To verify the data, process the data in the same way as described above. The resulting remainder should be 0 once all the data is complete.

Note that one potential issue is if the length of the message being processed is not a whole number multiple of S. In this case, the sender and receiver will need to agree on how to pad the message, which includes not only the values to pad it with, but also how far to pad it. For instance, if one side is using $S = 5$ and the other is using $S = 6$, they will need to ensure that the padded message length is a multiple of both 5 and 6. Alternatively, they could agree to always process the message in it's exact length, in which case the implementations will need to specially handle the case where the length is not a multiple of S. For instance, it could process as many S-bit blocks as possible, and then proceed to process any remaining bits using the standard "naive" approach.

In many cases, data will be a whole number of bytes and implementations will work in such a way that data is processed one byte at a time, avoiding any padding or special processing.

4 Table Modification

Recall that before computing the checksum, we need to append W bits to the message (on the sender side). This is equivalent to shifting in an extra W bits that are all 0 at the end of the calculation and processing them.

However, just as we saw in deriving the table based algorithm, shifting in K bits has the effect that the top K bits already in the register will produce a specific "constant" value which is a function only of those K bits and the generator polynomial G. This value is then XOR'd with the register.

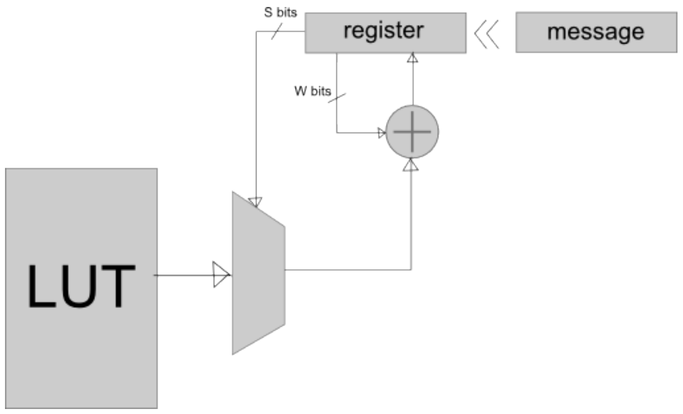
In the case of extending the message, K is W, which means it is all of the bits in the register that are shifted out and produce the value to be XOR'd. That means that when we do the XOR, the entire register is 0's, and so the XOR has the effect of putting the computed value directly into the register unmodified.

In other words, shifting the final W zeroes into the algorithm has the effect of forcing the last W message bits all the way through the algorithm.

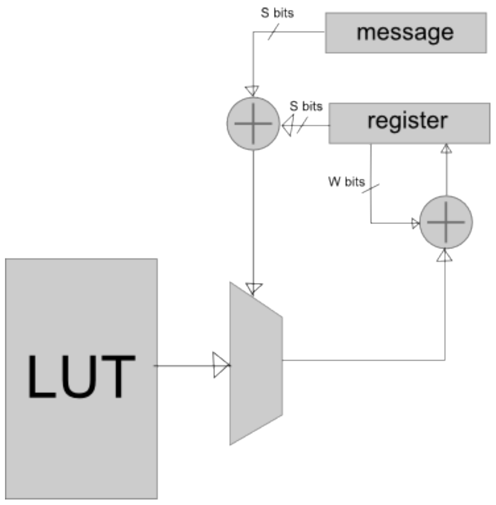
What that means is that message bits never actually need to be in the register; their only purpose is to control how the generator polynomial gets shifted and XOR'd onto itself (if you look back over the various implementations, particularly the long division exercise, you'll notice this is all that ever actually happens: the generator polynomial is shifted and then either XOR'd or not into itself, or into the results of a previous such operation).

We can therefore modify the algorithm. Instead of shifting data into the register only to have it XOR'd with some values then shifted out to select a value from T_G, we can just XOR the next S bits of the message with the top S bits from the register as they are shifted out, and use that to select from T_G.

So it used to look like this:



Now it looks like this:



In this way, message data is processed by the algorithm directly, instead of needing to be shifted through the register. This also allows up to skip the final W padding bits, which is the whole point of this modification: skip the last W bit-rounds.

5 Initial Values

We saw in the last section that as message bits shift out of the register (in the straight forward algorithm), they construct the final value by determining whether or not the polynomial G is XOR'd into the register.

But we didn't discuess what is supposed to happen before any data bits reach the top of the register. This is the first W bits of the algorithm.

The answer is thatis depends on the initial value of the register. In the pure polynomial division interpretation of CRC, the register is initially 0, which means as the first W bits of the message data are shifted in, W bits with value 0 are being shifted out. Since they are 0, we don't XOR the polynomial into the register, so these first W rounds don't do anything except shift the message data in. We've already gotten around the necessity for doing this with the modification to the table implementation, in which the message bits are XOR'd directly with the top S bits of the register to select an index, and never actually go into the register at all.

However, in real world applications, the reigster is often preloaded with some non-zero bits, which means the first W bit-rounds are actually important. (Preloading the register acts to offset the data and can improve certain aspects of error detection). However, as we've seen before with the table implementation, the results of performing some K number of rounds with K known bits in the top of the register can be expressed as XORing a bitstring which is dependent on the message data, with a bitstring which is only dependent on those K known bits (this is expressed in equote 1). In otherwords we can figure out the result that these initial W bits will produce in the same way we calculated the entries for the table (by preloading them into the register and then processing the algorithm through W bit rounds, with 0's for all message bits). Once we have this value (which is independent of the message data), we can simply initialize the register value with this result, instead of the original W-bit value.

The "original" initial value, the one used in the straight-forward algorithm, is called the *indirect* initial value. The modified value that we derive from this, which can be used in the modified table-based algorithm, is called the *direct* initial value. If you are using the straight-forward algorithm, (in which message data is shifted into the remainder register) you have to use the indirect initial value. If you're using the modified table implementation, you have to use the direct initial value.

As described above, converting the indirect to the direct is simply a matter feeding the indirect initial value through the straight forward algorithm. In other words, load the register with the indirect value, shift out 1 bit at a time and XOR the polynomial G into the register if and only if the shifted out bit is 1. This, of course, is just calculating the CRC of the indirect value (i.e., it is the remainder when you get when you multiply the indirect value by 2^W and then divide by the polynomial G.

If you wanted to convert from direct to indirect, you would need to find dividend which produces the direct value as a remainder when divided by the polynomial. This dividend would have to end in W bits that are all 0. I'm not sure if there's a way to do that or not.